**Experiment: 2**

**Extreme Programming (XP)**

Extreme Programming (XP), the most widely used approach to agile software development, emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

XP Values Beck defines a set of five values that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

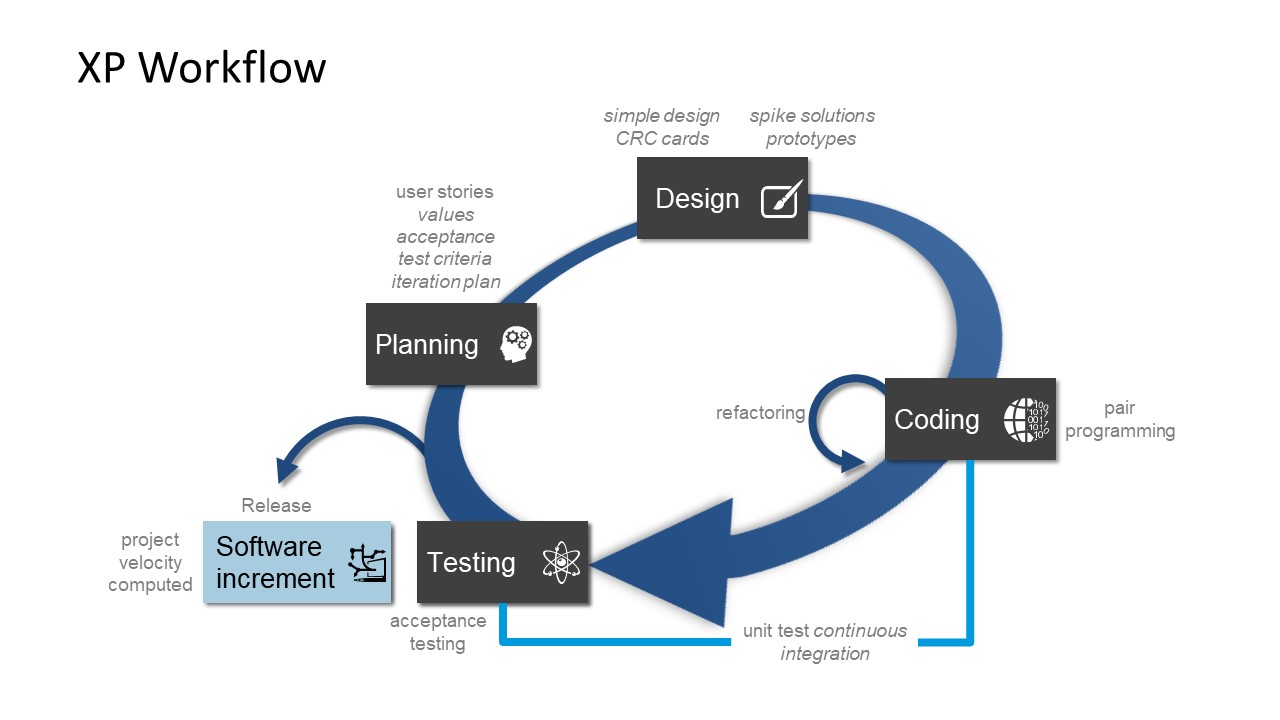
In order to achieve effective **communication** between software engineers and other stakeholders, XP emphasizes close, yet informal collaboration between customers and developers, the establishment of effective metaphors3 for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

To achieve **simplicity**, XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be refactored at a later time.

**Feedback** is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy the software provides the agile team with feedback. XP makes use of the unit test as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality.

Beck argues that strict adherence to certain XP practices demands **courage**. A better word might be discipline. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates **respect** among it members, between other stakeholders and team members, and indirectly, for the software itself. As they www.Jntufastupdates.com 6 achieve successful delivery of software increments, the team develops growing respect for the XP process.



**The XP Process Extreme Programming** uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are Fig : The Extreme Programming process

**• Planning.** The planning activity (also called the planning game) begins with listening—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.

**• Design**. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides www.Jntufastupdates.com 7 implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a spike solution, the design prototype is implemented and evaluated. XP encourages refactoring—a construction technique that is also a method for design optimization. Fowler describes refactoring in the following manner: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [that minimizes the chances of introducing bugs].

• **Coding.** After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers. A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance.

**• Testing.** The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages a regression testing strategy whenever code is modified. As the individual unit tests are organized into a “universal testing suite” integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: “Fixing small problems every few hours takes less time than fixing huge problems just before the deadline.”

**XP acceptance tests**, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

### Test-First Programming

Instead of following the normal path of:

develop code -> write tests -> run tests

The practice of [Test-First Programming](https://www.agilealliance.org/glossary/tdd/) follows the path of:

Write failing automated test -> Run failing test -> develop code to make test pass -> run test -> repeat

As with Continuous Integration, Test-First Programming reduces the feedback cycle for developers to identify and resolve issues, thereby decreasing the number of bugs that get introduced into production.

**Refactoring in Agile**

Refactoring is the practice of continuously improving the design of existing code, without changing the fundamental behavior. In Agile, teams maintain and enhance their code on an incremental basis from Sprint to Sprint. If code is not refactored in an Agile project, it will result in poor code quality, such as unhealthy dependencies between classes or packages, improper class responsibility allocation, too many responsibilities per method or class, duplicate code, and a variety of other types of confusion and clutter. Refactoring helps to remove this chaos and simplifies the unclear and complex code.

## 

## ****Implementation of Refactoring Techniques:****

There are multiple refactoring techniques available that will make the existing code better in terms of performance and maintainability. The basic purpose of refactoring in Agile or any other methodology is “Leave a module in a better state than you found it”

The Refactoring techniques are categorized by Martin Fowler as follows:

* Composing methods
* Moving features within Objects
* Organizing data
* Simplifying Conditional Expression
* Making method calls simpler

### Test cases for automation

If a manual test doesn't fit the bill, there is always the automation route. Before engineers develop test cases for automation, it's important to select the right test cases. Testers can employ test optimization to choose cases that provide the most coverage and ensure that their automated suite handles the highest test coverage market against the fewest number of tests.

Another element to consider is cost. Testers should select only reusable test cases because automation -- particularly at the GUI level -- requires costly maintenance.

### How to write test cases for automation

Automated test case writing is a complex endeavor that also requires a slightly different approach than its manual counterpart. Test cases for automation focus on the application state transitions and data changes, which makes it even more important that each test case contain only one test.

Test cases for automation must also further break down workflows compared to manual test cases. Templates for test cases for automation vary widely based on the automation tools in use, but they should all have the following components:

* **Preconditions or specifications**, detail how to get the application into the correct state for executing the test, including browser launch, logins, etc.;
* **Sync and wait statements**, allow the necessary time to get the application to the required state;
* **Test steps**, including data entry requirements, detailed steps how to reach the next required state and steps to return the application to its original state before test runs;
* **Comments**, explain the automation approach;
* **Debugging statements**; and
* **Output statements** describing where to record the results.

Both manual and automated testing are critical parts of an overall test strategy. Although both manual and automated test cases serve to verify functionality, they go about it in different ways and require different test case development.

It's important that testers understand the differences and similarities on how to write test cases for manual and automated situations to create [effective test cases](https://www.techtarget.com/searchsoftwarequality/tip/Efficient-test-case-design-techniques-to-boost-coverage) for all situations.